THE DATA RECONFIGURATION SERVICE--AN EXPERIMENT

IN ADAPTABLE, PROCESS/PROCESS COMMUNICATION

R. H. Anderson, V. Cerf, E. F. Harslem
J. F. Heafner, J. Madden, B. Metcalfe
A. Shoshani, J. White, D. Wood

July 1971

P-4673

THE DATA RECONFIGURATION SERVICE--AN EXPERIMENT

IN ADAPTABLE, PROCESS/PROCESS COMMUNICATION[*]

R. H. Anderson, V. Cerf, E. F. Harslem
J. F. Heafner, J. Madden, B. Metcalfe
A. Shoshani, J. White, D. Wood

The Rand Corporation, Santa Monica, California

I. INTRODUCTION

THE ARPA NETWORK AND SOME OF ITS GOALS

The nationwide ARPA Network [1-5] is composed of different host computers at geographically separated sites that are interconnected by small, standardized computers (IMPs) [6-7] and 50K-bit communication lines leased from the common carriers. The IMPs use store-and-forward switching to pass messages among hosts. Host computers vary in make, model, size, speed, and other hardware and software features. The Network is distributed and traffic routing is governed adaptively by the IMPs over redundant Network paths. Each participant can reliably access such various remote resources as programs, data, and unique hardware facilities. Individual programs at the sites control information flow.

Of primary concern are the fundamental intercommunication problems inherent in the marriage of autonomous hardware and software. No attempt has been made to provide compatible equipment in order to transfer, for example, large programs as a means of resource sharing.

One goal is to discover and validate techniques permitting uniform and easy access to all available resources, independent of hardware and software dissimilarities. More specifically, remote services should be as easily accessible as local ones, without noticeably degrading overall performance. Another goal is to allow more flexibility in the use of programming languages; because services will be offered remotely, compatible languages allowing program transferability are not required.

Such a network has many uses. Of greatest interest, however, are those that readily allow exploration of communication methods among different systems. One such generic use is *program sharing*, in which data are transmitted to a remote program and results are returned. Another is *data sharing*, in which small programs or algorithms are transmitted to operate on a large, remotely located data base.

EXAMPLES OF PROCESS INTERFACE DISPARITIES

The numerous instances of program and data sharing range from file transmission and data management systems to program/terminal coupling to a remote service. For example, weather modeling programs will run on the ILLIAC IV using parameters transmitted from Rand; results will be returned and reconfigured for graphical display and analysis. Although some of these programs exist today, their Network and graphical interfaces do not. Several remote job entry systems are now available on the Network (at UCSB and UCLA), yet minimal changes were made to those systems and thus their data input/output (I/O) formats differ considerably. At MIT, the special Evans and Sutherland graphic hardware[+] is offered as a remote service. It is desirable to use this service from such various kinds of graphics terminals as the IMLAC and ARDS.

To further amplify the problem of different software interfaces, many sites will have a minimal host configuration that will restrict their data reformatting capabilities, but that should not restrict their access to remote resources requiring different formats.

_____

[+]Evans & Sutherland Computer Corporation, 3 Research Road, Salt Lake City, Utah 84112.

Examining the currently proposed and existing services, the kinds
of data manipulations most frequently encountered are: character set
conversions, prefacing and stripping leaders of messages, packing and
unpacking repeated symbol strings, generating message counters and
flags to be inserted into the data stream, graphic device code conver-
sions, data field-transposition, and reformatting files.

This paper discusses one recent approach for providing the above
kinds of data transformations in a way that is transparent to the
terminals and programs involved.

THE DATA RECONFIGURATION SERVICE (DRS) APPROACH

Application programs require specific I/O data formats that differ
from program to program. One approach recently adopted for providing
resource sharing of disparate programs is to develop specific dialogs
for classes of programs. Each such program must then be retrofitted
with one of the standard dialog interfaces. The DRS exhibits a dif-
ferent view of coupling variegated processes and terminals. The pre-
mise underlying DRS is that the Network should adapt to the individual
program requirements rather than changing each program to comply with
a standard. This position does not preclude the use of standards that
describe the formats of Network message contents; it is merely an in-
terpretation of a standard as being a desirable mode of operation, but
not a necessary one.

In addition to differing program requirements, a format mismatch
occurs when users wish to employ many different kinds of consoles to
attach to a single remote service program. It is likewise desirable
to have the Network adapt to individual console configurations rather
than requiring unique software packages for each console transformation.

One approach to providing adaptation is for those sites with sub-
stantial computing power to offer a data reconfiguration service; this
paper describes such a service, the DRS, currently being implemented
at MIT, UCLA, UCSB, and The Rand Corporation. The University of Illinois,
MITRE, and others will experiment with its use.

The envisioned *modus operandi* of the service is that an applications
programmer defines *forms* that describe data reconfigurations. The service

stores the forms by name. At a later time (or immediately thereafter),
a user (perhaps a non-programmer) employs the service to accomplish a
particular transformation of a Network data stream passing between a
using process and a serving process. He accomplishes this by calling
the form by name and identifying it with the using and serving processes.

The DRS attempts to provide a notation for form definition tail-
ored to some specifically needed instances of data reformatting. At
the same time, the DRS keeps the notation and its underlying implemen-
tation within some utility range that is bounded on the lower end by
a notation expressive enough to make the experimental service useful,
and bounded on the upper end by a notation short of a general-purpose
programming language.

II. OVERVIEW OF THE DATA RECONFIGURATION SERVICE

ELEMENTS OF THE DATA RECONFIGURATION SERVICE

An implementation of the DRS includes a module for Network connection protocols to establish logical message paths between the end processes that wish to pass data. It also includes a module (the Form Machine) to accept and apply the definitions of data reconfigurations (forms). Lastly, a file management module exists for saving and retrieving forms.

This section highlights connections and requests. Section III details the Form Machine language. File storage is not described in this paper because it is transparent to the user and its implementation is different at each DRS host.

NETWORK CONNECTIONS

There are three kinds of Network connections to the DRS (see Fig. 1).

1. The control connection (CC) is between an originating user and the DRS. It is instigated by the user to define forms and to request the user connection (UC) and the server connection (SC), along with the application of form(s) to data passing between UC and SC.
2. The UC is between a user process and the DRS. It is established by the DRS at the request of the originating user.
3. The SC is between the DRS and the serving process. It, too, is established by the DRS at the request of the originating user.

The user process behaves as if it were connected directly to the server process, and vice versa. The DRS appears transparent to both processes; its function is to reconfigure data that pass in each direction between them into formats amenable to each of their processing requirements. Because the goal is to adapt the Network to user and server processes, minimal requirements are imposed on the UC and SC.
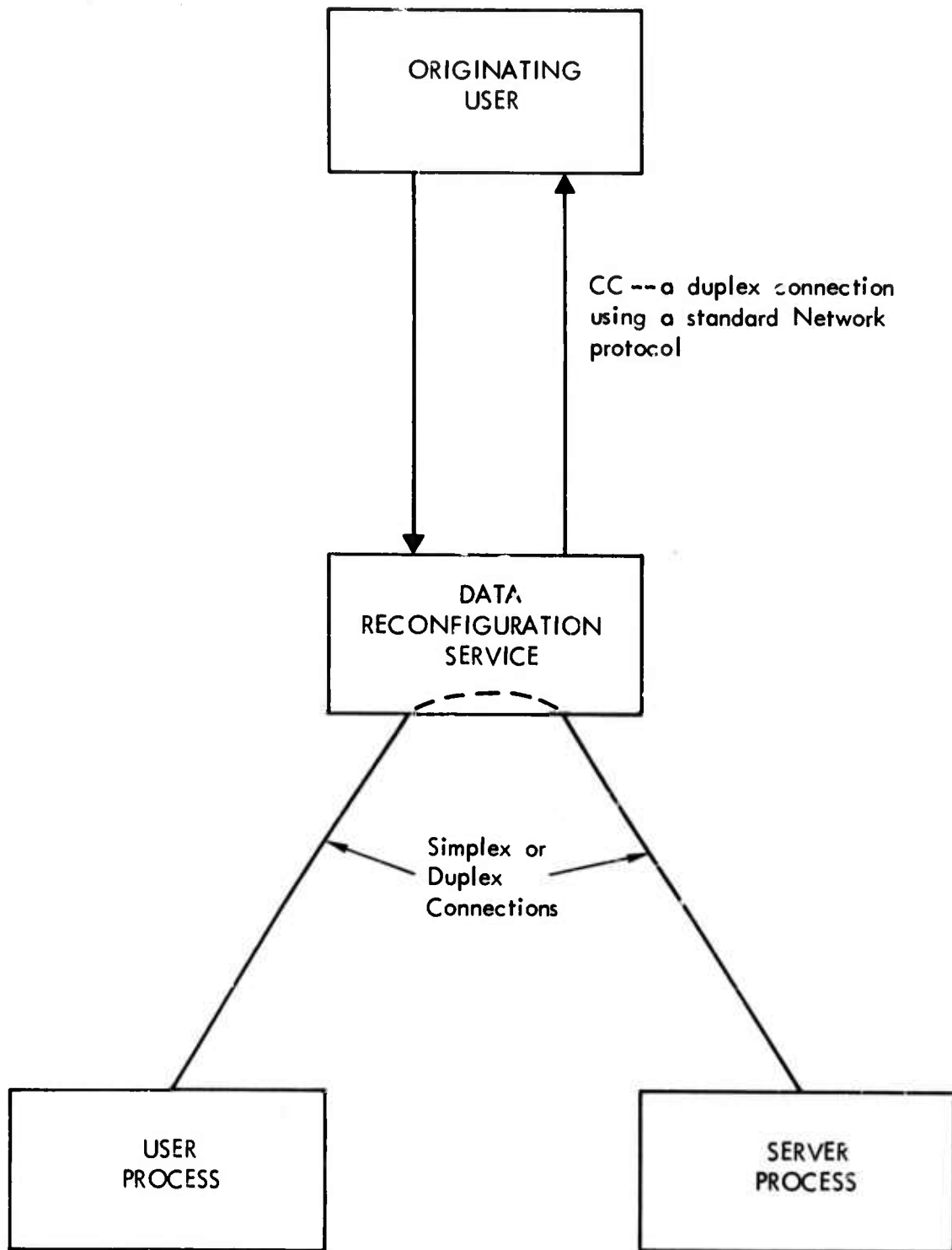
Fig. 1--DRS Network Connections

REQUESTS OVER THE CONTROL CONNECTION

Over a control connection, the dialog is directly between an originating user and the DRS, where the user defines forms or assigns predefined forms to connections for reformatting. Messages sent over a control connection are formatted according to a Network standard.

When an originating user connects to DRS, he supplies an identifier as a qualifier to guarantee uniqueness of his form names. The user can request the following operations:

1. Accept a form definition;
2. Purge a form definition;
3. List qualified form names;
4. List the source text of a form;
5. Make a simplex or duplex logical connection between a user and a server process. The connection can be made in several ways, i.e., with or without a Network standard connection protocol;
6. Abort a user/server connection.

When a user/server connection is severed either by the processes themselves or by an abort request, the DRS sends an appropriate return code to the originating user.

III. THE FORM MACHINE

I/O STREAMS AND FORMS

This section describes the syntax and semantics of forms that
specify the data reconfigurations. The Form Machine gets an input
stream, reformats the input stream according to a form describing
the reconfiguration, and emits the reformatted data as an output
stream.

It is helpful to envision the application of a form to the data
stream, depicted in Fig. 2. An input stream pointer identifies the
position of data (in the input stream) that is being analyzed, at any
given time, by a part of the form. Likewise, an output stream pointer
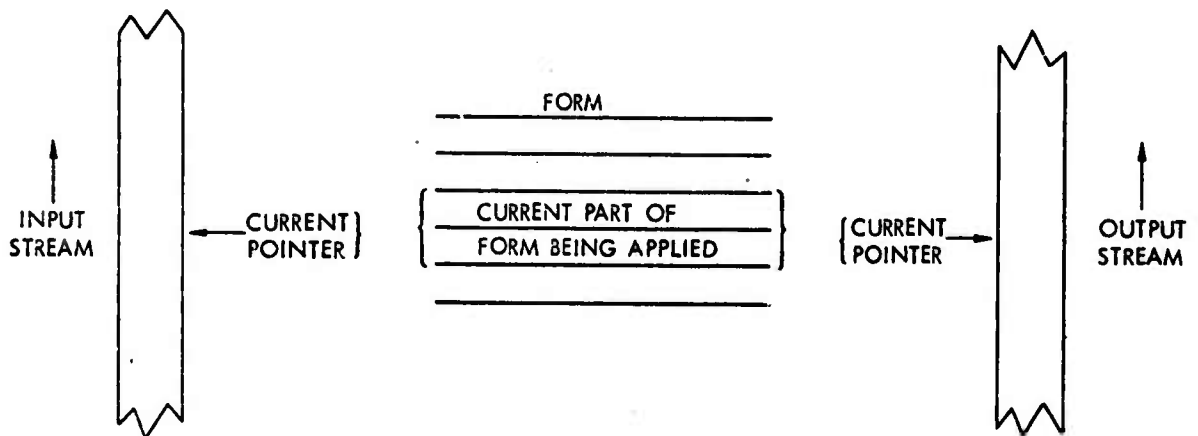locates data emitted in the output stream.

Fig. 2--Application of Form to Data Streams

FORM MACHINE SYNTAX[†]

form	::=	$\{rule\}_1^{\infty}$
rule	::=	$\{INTEGER\}_0^1$ $\{terms\}_0^1$ $\{:terms\}_0^1$;
terms	::=	term $\{,term\}_0^{\infty}$
term	::=	identifier $\mid$ $\{identifier\}_0^1$ descriptor $\mid$ comparator
descriptor	::=	$(\{replicationexpr\}_0^1$, datatype , $\{concatexpr\}_0^1$, $\{arithmeticexpr\}_0^1$ $\{:options\}_0^1)$
comparator	::=	(concatexpr connective concatexpr $\{:options\}_0^1)$ $\mid$ (identifier •<=• concatexpr $\{:options\}_0^1)$
connective	::=	.LE. $\mid$ .LT. $\mid$ .GE. $\mid$ .GT. $\mid$ .EQ. $\mid$ .NE.
replicationexpr	::=	# $\mid$ arithmeticexpr
datatype	::=	B $\mid$ O $\mid$ X $\mid$ E $\mid$ A $\mid$ ED $\mid$ AD $\mid$ SB $\mid$ T(identifier)
concatexpr	::=	value$\mid$ $\{\mid\mid$ value$\}_0^{\infty}$
value	::=	literal $\mid$ arithmeticexpr
arithmeticexpr	::=	primary $\{operator$ primary$\}_0^{\infty}$
operator	::=	+ $\mid$ - $\mid$ * $\mid$ /
primary	::=	identifier $\mid$ L(identifier) $\mid$ V(identifier) $\mid$ INTEGER
literal	::=	literaltype "$\{CHARACTER\}_0^{256}$ "
literaltype	::=	B $\mid$ O $\mid$ X $\mid$ A $\mid$ E $\mid$ ED $\mid$ AD $\mid$ SB
options	::=	sfur (arithmeticexpr) $\mid$ sfur (arithmeticexpr) , sfur (arithmeticexpr)
sfur	::=	S $\mid$ F $\mid$ U $\mid$ SR $\mid$ FR $\mid$ UR
identifier	::=	ALPHABETIC $\{ALPHAMERIC\}_0^3$

[†]These syntactic statements are referred to in the following semantic descriptions.

FORMS

A form is an ordered set of rules.

$$\text{form} \quad ::= \quad \{\text{rule}\}_1^\infty$$

The current rule is applied to the current position of the input stream. If the rule fails to correctly describe the current input then another rule is made current and applied to the input stream.[†] The next rule to be made current is either explicitly specified by the current term in the current rule or it is the next sequential rule by default.

If the current input stream is correctly described, then some data may be emitted at the current position in the output stream according to the rule. The input and output stream pointers are advanced over the described and emitted data, respectively; the next rule is applied to the now current position of the input stream.

Application of the form is terminated when an explicit return, e.g., UR (arithmetic expression) is encountered in a rule. The user and server connections are closed and the evaluated return code (arithmetic expression) is sent to the originating user.

RULES

A rule is a replacement, comparison, and/or an assignment operation of the form shown below.

$$\text{rule} \quad ::= \quad \{\text{INTEGER}\}_0^1 \ \{\text{terms}\}_0^1 \ \{:\text{terms}\}_0^1 \ ;$$

The optional integer (rule name) exists so that the rule may be referenced elsewhere in the form for explicit rule transfer of control. Integers are in the range $0 \geq \text{INTEGER} \geq 9999$. Rules need not be named in ascending numerical order.

[†]If only a part of the rule succeeds, the input pointer is not advanced.

TERMS

The input stream is described by zero or more terms,

$$\{terms\}_0^1$$

and the output stream is described by zero or more terms,

$$\{:terms\}_0^1$$

where

$$terms ::= term \{,term\}_0^\infty \ .$$

Terms are expressed in one of the formats indicated below.

$$term ::= identifier \mid \{identifier\}_0^1 \ descriptor \mid comparator$$

Term Format 1

The first term format, *identifier*, is a symbolic reference to a previously identified term (term format, 2 below) in the form. It takes on the same attributes (replication, type, value, length) as the term by that name and is normally used to emit data.

Term Format 2

The second term format, $\{identifier\}_0^1$ descriptor, is used to collect input or to emit.

$$descriptor ::= (\{replicationexpr\}_0^1 \ , \ datatype \ , \ \{concatexpr\}_0^1 \ ,$$
$$\{arithmeticexpr\}_0^1 \ \{:options\}_0^1)$$

The above five descriptor elements[†] correspond to the attributes replication, data type, value, length, and transfer of control, respectively.

[†]See the IBM System Reference Library Form C28-6514 for a similar interpretation of the pseudo instruction, Define Constant, after which the descriptor was modeled.

The replicationexpr, if specified, causes the unit value of the term to be repeated the number of times indicated by the replication expression's value. The unit value of the term (quantity to be replicated) is determined from the composite of data type, value expression, and length expression attributes. The data type defines the kind of data being specified. The value expression specifies a nominal value that is augmented by the other term attributes. The length expression determines the unit length of the term.

The terminal symbol # in a replication expression means an arbitrary replication factor. It is explicitly terminated by a non-match to the input stream. Termination may result from exceeding the 256-character limit.

A null replication expression has a default value of one. Arithmetic expressions are evaluated from left-to-right with no precedence.

The L(identifier) is a *length operator* that generates a 32-bit binary integer corresponding to the length of the term named. The V(identifier) is a *value operator* that generates a 32-bit binary integer corresponding to the value of the term named. The T(identifier) is a *type operator* that generates a type-code for the term named.

The *data type* describes the kind of data that the term represents.[†]

Data Type	Meaning	Unit Length
B	Bit string	1 bit
O	Bit string	3 bits
X	Bit string	4 bits
E	EBCDIC character	8 bits
A	Network ASCII character	8 bits
AD	ASCII encoded decimal	8 bits
ED	EBCDIC encoded decimal	8 bits
SB	Signed binary	1 bit

The *value expression* is the nominal value of a term expressed in the format indicated by the data type. It is repeated according to the

---

[†]It is expected that such additional data types as floating-point and user-defined types will be added as needed.

replication expression. A null value expression in the input stream defaults to the data present in the input stream and generates padding in the output stream according to the restrictions and interpretations stated later. The input data must comply with the datatype attribute, however.

The *length expression* states the length of the field containing the nominal value. If the length expression is less than or equal to zero, the term succeeds but the appropriate stream pointer is not advanced. Positive lengths cause the appropriate stream pointer to be advanced if the term otherwise succeeds.

Options is defined under Term and Rule Sequencing.

Term Format 3

The third term format is used for assignment and comparison

$$\text{comparator} \; := \; (\text{concatexpr} \quad \text{connective} \quad \text{concatexpr} \; \{\text{:options}\}_0^1) \; | $$
$$(\text{identifier} \; \cdot \mathord{<}\mathord{=} \cdot \; \text{concatexpr} \; \{\text{:options}\}_0^1)$$

The assignment operator $\cdot \mathord{<}\mathord{=} \cdot$ assigns the value to the identifier. The connectives have their usual meanings. Values to be compared must have the same type and length attributes or an error condition arises and the form fails.

The Application of a Term

The elements of a term are applied by the following sequence of steps.

1. The data type (datatype), value expression (concatexpr), and length expression (arithmeticexpr) together specify a unit value, call it x.

2. The replication expression (replicationexpr) specifies the number of times x is to be repeated. The value of the concatenated xs becomes y of length L.

3. If the term is an input stream term, then the value of y of length L is tested with the input value beginning at the current input pointer position.

4. If the input value satisfies the constraints of y over length L, then the input value of length L becomes the value of the term.

In an output stream term, the procedure is the same except that the source of input is the value of the term(s) named in the value expression and the data is emitted in the output stream.

The above procedure is modified to include a one term look-ahead where replicated values are of indefinite length because of the arbitrary symbol, #.

Restrictions and Interpretations of Term Functions

1. Terms having indefinite lengths because their values are repeated according to the # symbol, must be separated by some type-specified data, such as a literal.[†]

2. Truncation and padding is:

   a. Character to character (A ↔ E) conversion is left-justified and truncated or padded on the right with blanks.

   b. Character to numeric and numeric to numeric conversations are right-justified and truncated or padded on the left with zeros.

   c. Numeric to character conversation is right-justified and left-padded with blanks.

3. The following are ignored in a form definition over the control connection:

   a. Control characters.

   b. Blanks except within quotes.

   c. /* string */ is treated as comments except within quotes.

4. The following defaults prevail where the term part is omitted:

   a. The replication expression defaults to one.

   b. # in an output stream term defaults to one.

[†] A literal is not specifically required, however. An arbitrary number of ASCII characters could be terminated by a non-ASCII character.

    c. The value expression of an input stream term defaults to the value found in the input stream, but the input stream must conform to data type and length expression. The value expression of an output stream term defaults to padding only.

    d. The length expression defaults to the size of the quantity determined by the data type and value expression.

    e. Control defaults to the next sequential term if a term is successfully applied; otherwise, control defaults to the next sequential rule.

5. Arithmetic expressions are evaluated left-to-right with no precedence.

6. The following limits prevail:

    a. Binary lengths are $\leq$ 32 bits.

    b. Character strings are $\leq$ 256 8-bit characters.

    c. Identifier names are $\leq$ 4 characters.

    d. Maximum number of identifiers is $\leq$ 256.

    e. Label integers are $\geq$ 0 and $\leq$ 9999.

7. Value and length *operators* produce 32-bit binary integers. The value operator is currently intended for converting A or E type decimal character strings to their binary correspondants. For example, the value of E'12' would be 0......01100. The value of E'AB' would cause the form to fail.

TERM AND RULE SEQUENCING

    Rule sequencing may be explicitly controlled by using

$$\{:\text{options}\}_0^1 ,$$

defined as

    options ::= sfur(arithmeticexpr) |

                  sfur(arithmeticexpr) , sfur(arithmeticexpr)

    sfur ::= S | F | U | SR | FR | UR

respectively. The arithmetic expression evaluates to an integer; thus, transfer can be effected from within a rule (at the end of a term) to the beginning of another rule. R means terminate the form and return the evaluated expression to the initiator over the control connection.

If terms are not explicitly sequenced, the following defaults prevail:

1. When a term fails, go to the next sequential rule.
2. When a term succeeds, go to the next sequential term within the rule.
3. At the end of a rule, go to the next sequential rule.

In the following example, note the correlation between transfer of control and movement of the input pointer.

```
1    XYZ(,B,,8:S(2),F(3)) : XYZ ;
2    . . . . . . .
3    . . . . . . .
```

The value of XYZ will never be emitted in the output stream because control is transferred out of the rule upon either success or failure. If the term succeeds, the 8 bits of input are assigned as the value of XYZ and rule 2 is then applied to the same input stream data. That is, because the complete left hand side of rule 1 was not successfully applied, the input stream pointer is *not* advanced.

IV. EXAMPLES

The following examples (forms and also single rules) are simple representative uses of the Form Machine.

FIELD INSERTION

To insert a field, separate the input into the two terms to allow the inserted field between them. For example, if the input stream contained pairs of numbers encoded as ASCII, separated by a slash (i.e., 123/456/...), the following form labels them as x, y pairs separated by a line feed, carriage return (i.e., X=123/Y=456 (LF) (CR) ...)

```
1    XVAL (#,A,,1), (,A,A"/",1),YVAL (#,A,,1),(,A,A"/",1)
     /*pick up the x as XVAL and y as YVAL */

2    : (,A,A"X=",2),XVAL,(,A,"/Y=",3),YVAL
     /*emit the labels followed by the values of x, y */

3    : (,X,X"OAOD" ,4: U(1))
     /*emit the line feed, carriage return and loop back for the
     next pair */
```

DELETION

Data to be deleted should be isolated as separate terms on the left in order to be omitted (by not emitting them) on the right.

```
(,B,,8),              /*isolate 8 bits to ignore*/

SAVE(,A,,10)          /*extract 10 ASCII characters from
                        input stream*/

:(,E,SAVE,);          /*emit the characters in SAVE as
                        EBCDIC characters whose length
                        defaults to the length of SAVE
                        (i.e., 10), and advance to the
                        next rule*/
```

In the above example, if either input stream term fails, the next sequential rule is applied.

VARIABLE LENGTH RECORDS

Some devices, terminals, and programs generate variable length records. The following rule picks up variable length EBCDIC records and translates them to ASCII.

CHAR(#,E,,1),	/*pick up all (an arbitrary number of) legal EBCDIC characters in the input stream*/
(,X,X"FF",2)	/*followed by a hexadecimal literal, FF (terminal signal)*/
:(,A,CHAR,),	/*emit them as ASCII*/
(,X,X"OD",2);	/*emit an ASCII carriage return*/

STRING LENGTH COMPUTATION

It is often necessary to prefix a length field to an arbitrarily long character string. The following rule prefixes an EBCDIC string with a one-byte length field.

Q(#,E,,1),	/*pick up all legal EBCDIC characters*/
TS(,X,X"FF",2)	/*followed by a hexadecimal literal, FF*/
:(,B,L(Q)+2,8),	/*emit the length of the characters plus the length of the literal plus the length of the count field itself, in an 8-bit field*/
Q,	*/emit the characters */
TS;	*/emit the terminal*/

TRANSPOSITION

It is often desirable to reorder fields, such as the following example.

Q(,E,,20), R(,E,,10) , S(,E,,15), T(,E,,5) : R, T, S, Q;

The terms are emitted in a different order.

CHARACTER PACKING AND UNPACKING

In systems such as HASP, repeated sequences of characters are packed into a count followed by the character, for more efficient storage and transmission. The first form packs multiple characters and the second unpacks them.

```
/*form to pack EBCDIC streams*/

/*returns 99 if OK, input exhausted*/

/*look for terminal signal FF which is not a legal EBCDIC*/

/*duplication count must be 0-254*/

1 (,X,X"FF",2 : SR(99)) ;

/*pick up an EBCDIC char/*

CHAR(,E,,1) ;

/*get identical EBCDIC chars/*

LEN(#,E,CHAR,1)

/*emit the count and the char/*

: (,B,L(LEN)+1,8), CHAR, (:U(1));

/*end of form*/
```

```
/*form to unpack EBCDIC streams*/

/*look for terminal*/

1 (,X,X"FF",2 : SR(99)) ;

/*emit character the number of times indicated*/

/*by the count, in a field the length indicated*/

/*by the counter contents*/

CNT(,B,,8), CHAR(,E,,1) : (CNT,E,CHAR,1:U(1));

/*failure of form*/

(:UR(98))
```

REFERENCES

1. Roberts, L. G., and B. D. Wessler, "Computer Network Development to Achieve Resource Sharing," *AFIPS Conference Proceedings*, Vol. 36, 1970, pp. 543-549.

2. Heart, F. E., R. E. Kahn, S. M. Ornstein, W. R. Crowther, and D. C. Walden, "The Interface Message Processor for the ARPA Computer Network," *AFIPS Conference Proceedings*, Vol. 36, 1970, pp. 551-567.

3. Kleinrock, L., "Analytic and Simulation Methods in Computer Network Design," *AFIPS Conference Proceedings*, Vol. 36, 1970, pp. 569-579.

4. Frank, H., I. T. Frisch, and W. Chou, "Topological Considerations in the Design of the ARPA Computer Network," *AFIPS Conference Proceedings*, Vol. 36, 1970, pp. 581-587.

5. Carr, C. S., S. D. Crocker, and V. G. Cerf, "HOST-HOST Communication Protocol in the ARPA Network," *AFIPS Conference Proceedings*, Vol. 36, 1970, pp. 589-597.

6. *Interface Message Processor: Operating Manual*, Bolt, Beranek and Newman, Inc., Report No. 1877, February 1970.

7. *Interface Message Processor: Specifications for the Interconnection of a HOST and an IMP*, Bolt, Beranek and Newman, Inc., Report No. 1822, October 1970.

BIBLIOGRAPHY

Baran, P., "On Distributed Communication Networks," *IEEE Transactions on Communication Systems*, Vol. CS-12, March 1964.

Marill, T., and L. G. Roberts, "Toward a Cooperative Network of Time-Shared Computers," *AFIPS Conference Proceedings*, Vol. 29, 1966, pp. 425-431.

Preceding page blank